Using artdaq to create a prototype event-builder for Darkside-50

The SSI and RSI groups [1]

1 Introduction

We have created a prototype event-builder for DarkSide-50, as a demonstration of how such a program might be built using the **artdaq** framework. **artdaq** provides the "generic" part of the event-building infrastructure. This generic infrastructure assumes that events are made up of one or more *fragments*, each of which can be uniquely identified by a pair of numbers: an *event id* (also called *sequence id*), which identifies the event to which the fragment belongs, and a *fragment id*, which indicates which piece of the event is represented by the fragment. For DarkSide-50, the fragment corresponds to the data from one digitizer board.

artdaq uses MPI[1] to create a program that runs multiple Unix processes, possibly on multiple nodes. It also uses OpenMP[2] for thread-level parallelism within some of these processes. It uses the **art** framework [3] to run *modules*, which would normally be supplied by an experiment, to do work. In the prototype, we have not separated the experiment-specific code into libraries separate from the **artdaq** framework code.

Our prototype event-builder program, *builder*, does not connect to a live data feed; it reads data from a configurable number of files, each file containing simulated data from one digitizer board. *builder* combines the fragments carrying the board data into complete events which are then sent to the be processed by the **art** event-processing framework. We have written a data compression module, suitable for use in **art**, that compresses each board using the Huffman compression algorithm[4], with a symbol table that is read at configuration time.

builder contains processes with several different tasks.

1. Each "detector" process emulates the data feed of one digitizer board; each reads one data file, and sends the fragments to its corresponding "fragment receiver" process. Our emulation uses MPI over InfiniBand to send data. In our emulation, the data as sent already include fragment and event ids, as well as timestamps. We rely on the fragment and event id information during routing of fragments and event building, and do not use the timestamp.

2. Each "fragment receiver" process receives fragments from a single "detector" and uses each fragment's event id to route the fragment to the correct "event builder" process.

3. Each "event builder" process collects all the fragments for the events routed to it.When the "event builder" process receives a fragment that completes an event, the completed event is then sent to another thread in the process for processing by **art**. The modules to be run by **art** are specified using **art**'s runtime configuration mechanism, and thus can be changed without recompilation of the program.

[1] Contacts: Kurt Biery (biery@fnal.gov), Jim Kowalkowski (jbk@fnal.gov) and Marc Paterno (paterno@fnal.gov).

We have written an **art** module that does (lossless) Huffman compression of the fragments carrying the board data. Within each event, each fragment is compressed independently of the other fragments of that event; they are compressed in threads running in parallel.

2 Methods, Assumptions, and Procedures

We performed our performance measurements on a set of four 32-core Linux nodes. Each node has 4 sockets, each with an 8-core AMD 6128 processor. Each node has 64 GiB RAM and a RAID I disk. They are running Scientific Linux Fermi v5.5. They are connected to a switch via QDR Infiniband and are on a private 1 Gb ethernet private network.

We built the software using the GCC C++ compiler, version 4.6.2, using *-O3* optimization. Table 1 lists the software products and versions used in these measurements. **art_openmp** is an early prototype version of an OpenMP-enabled **art** product. In this exercise, we have not made use of any of the thread parallelism portions of **art_openmp**; the only thread parallelism used is within the compression module.

product	version and qualifiers
artdaq	commit #3d76010e
art_openmp	v0.00.04 -q e1

Table 1: The software products and versions used in the measurements in this paper.

The configuration run used 5 “detectors”, 5 “fragment receivers”, and 5 “event builders”, distributed among the 4 nodes as shown in Figure 1. Each of the “event builder” processes

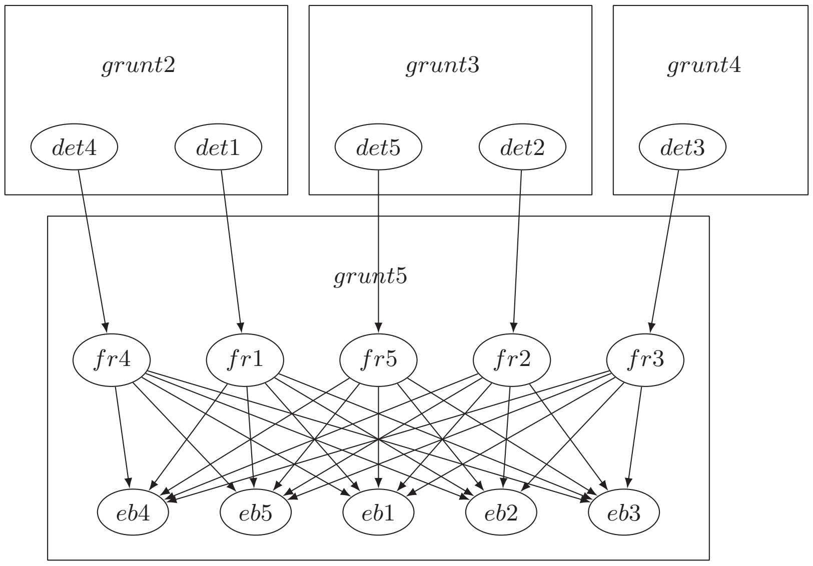

Figure 1: The layout of processes used in this experiment. *det* indicates a “detector” process, *fr* indicates a “fragment receiver” process, and *eb* indicates an “event buider” process. It is the “event builder” processes that run **art**.

was configured to use an OpenMP thread pool of size 5, and thus 5-way thread parallelism was enabled in each “event builder” process. The MPI program launch mechanism starts all the processes. Each “detector” process reads its own data file. They do not begin sending data to the “fragment receivers” until all have read the full data file. This is done so that the throughput measurements do not depend on the disk reading speed of the “detector” processes. After sending the last of the fragements read from its file, each “detector” processes sends and end-of-data marker to its “fragment receiver”.

Each “fragment receiver” process receives data from a single “detector”, looks at the event id in the data, and routes the fragment to the appropriate “event builder”, based on a round-robin algorithm. When a “fragment receiver” receives an end-of-data marker from its “detector”, it sends end-of-data markers to each “event builder”.

Each “event builder” process receives fragments from all “fragment receivers”. When an “event builder” has received all the fragments of a given event, it sends that event to the thread running **art**. **art** then passes the event to each of the configured modules. In our prototype, the only module used is the module that compresses the fragment data. Orderly program shutdown happens when the last of the “event builder” processes sees the last of the end-of-data markers it is expecting.

3 Results and Discussion

3.1 Timing results

Figure 2 shows the time taken by the compression module to compress each of the first 250 events in the data stream. The first event takes approximately 6 ms longer than each of the remaining events, but each of the compression modules seems to have reached a steady state by the second event it sees. This may indicate the time taken to create the OpenMP thread pool.

Compression of the fragments for one event takes on average about 20 ms (mean is 20.37 ms, standard deviation of 0.58 ms); this is the wall-clock time required for the team of 5 threads to compress the 5 fragment. This corresponds to an aggregate event throughput mean of 246 events/s, with a standard deviation of 7 events/s. Fig. 3 shows the distribution of measured compression times. It is clearly non-gaussian, and when the data are separated by the id of the “event builder” that processed the event, two approximately gaussian peaks are clearly evident.

3.2 Compression results

The Huffman algorithm we have coded takes, at configuration time, a symbol frequency table, so that it is not necessary to scan the data to obtain symbol frequencies. We constructed the symbol frequency table from the same data we are compressing. With this initial compression algorithm, we have achieved a mean compression factor of 4.87, with a standard deviation of 0.17. Figure 4 shows the distribution of compression ratios achieved in the data sample, on a fragment-by-fragment basis.

3.3 Throughput results

With the given configuration, we obtain throughput rates (measuring from the time after the “detectors” have read the data files, but before they being sending data, to the time at which the “event builder” processes have seen the last event) of 2–2.2 GiB/s. We have measured both the single-process throughput rate and the aggregate rate for all “event builder” processes. We measure the rate averaged over approximately one-second intervals. Figure 5 shows the single-

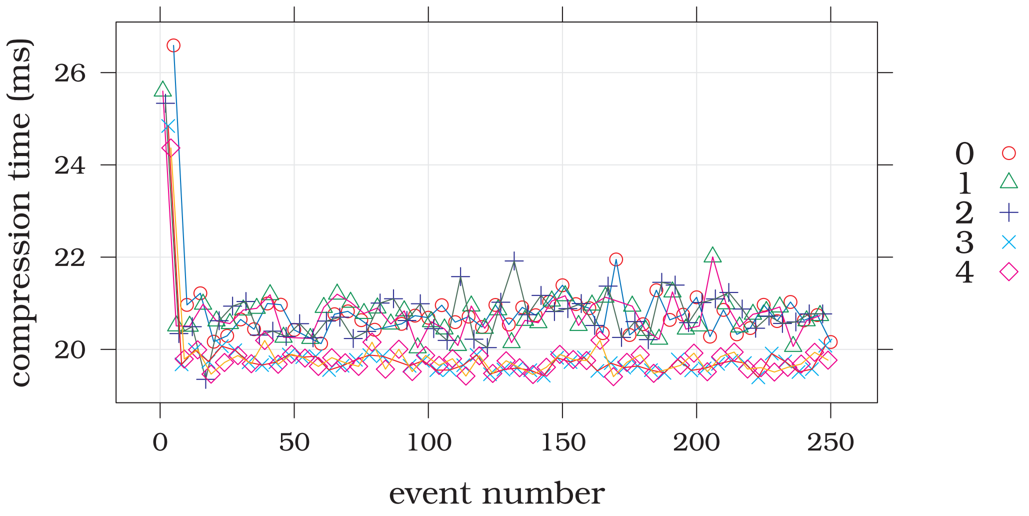

Figure 2: The time taken by the each of the compression modules on each of the first 250 events. There are 5 **art** processes used, each running one compression module. The different symbol colors and shapes denote the different **art** processes.

process throughput rate, and figure 6 shows the aggregate rate. The mean value over the duration of the run is 371 events/s, and the standard deviation of the distribution is 12 events/s.

4 Conclusions

We have not yet done any significant optimization of the code, and have several opportunities to take more advantage of the native parallelism of the problem. For example, we are currently using board-level parallelism in the compression module; it would be simple, and given sufficient computing resources faster, to use channel-level parallelism. This would give us up to 38-way, rather than 5-way, parallelism.

Even without optimizations, the system we prototyped is capable of processing approximately 250 event/s, with all the "fragment receiving" and "event building" processes running on a single 32-core commodity computer.

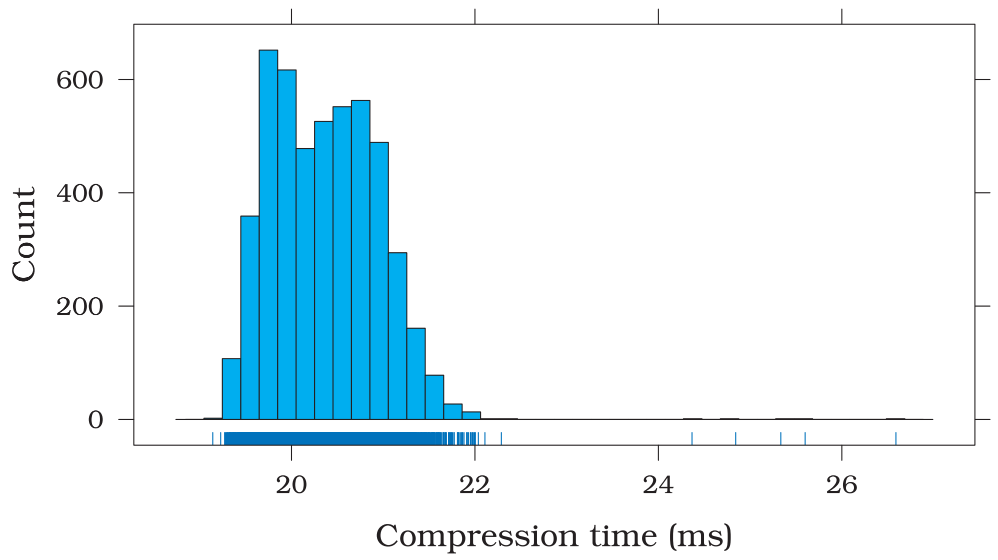

Figure 3: The distribution of the time taken for a Huffman compression module to compress an event. The algorithm has been trained on the same data sample being compressed in this measurement. Note that the “rug” plot below the histogram shows 5 outliers; these are the 5 events each of which is the first processed by an individual compression module.

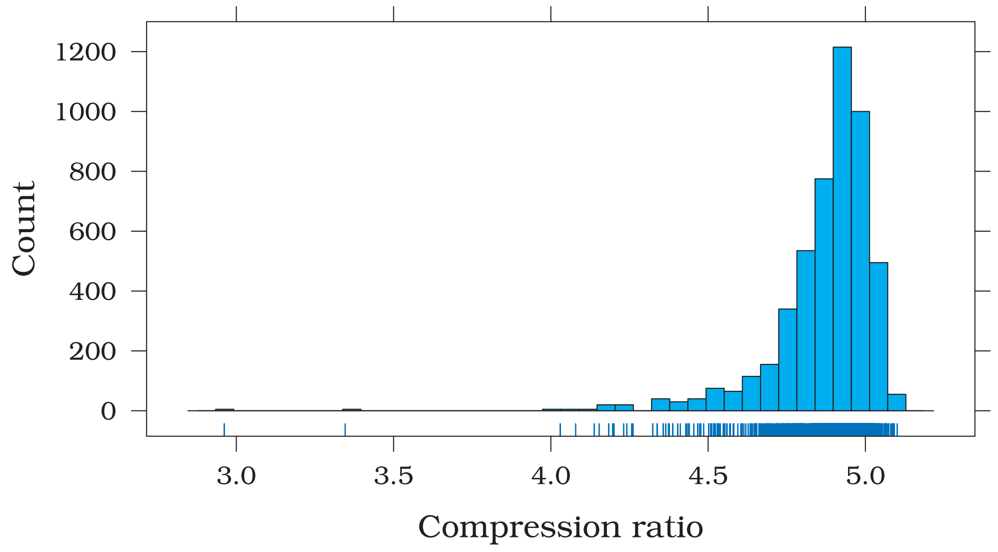

Figure 4: The distribution of the compression factor of the (lossless) Huffman compression algorithm, for each board in each event.

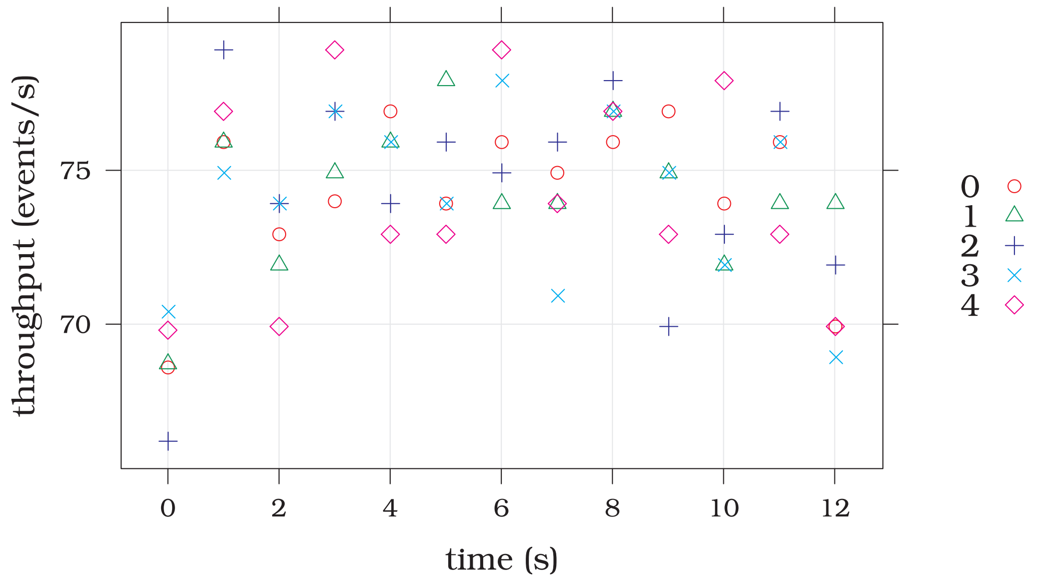

Figure 5: Throughput rate for each of the 5 “event builder” processes. The different symbol colors and shpaes denote the different **art** processes.

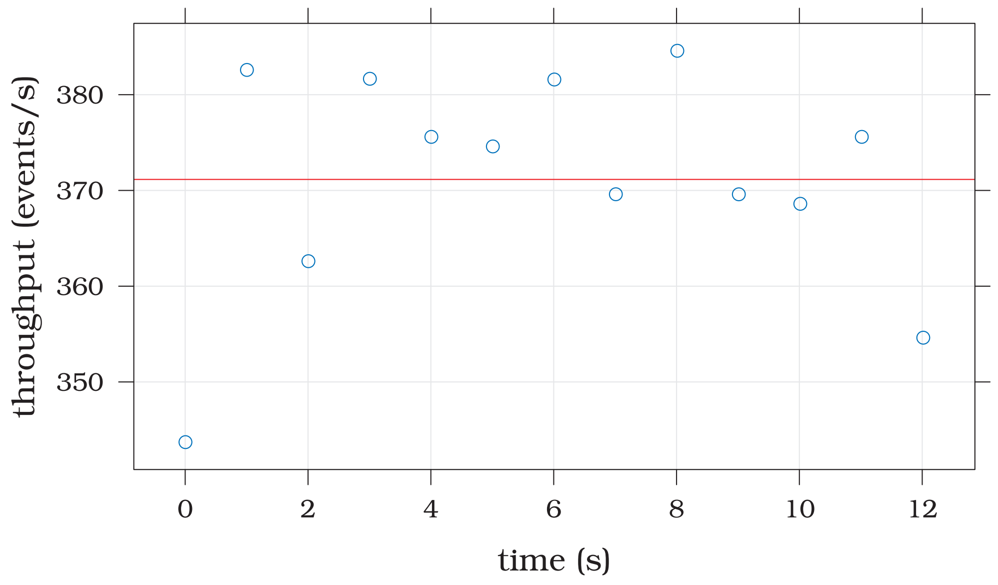

Figure 6: Aggregate throughput rate of the 5 “event builder” processes. The red line shows the mean value, averaged over the full time of the run.

Bibliography

[1] **The MPI-2 standard**, available at http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm.

[2] **OpenMP Application Program Interface**, version 3.1, July 2011, available at http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[3] Information about the **art** event processing framework is available at https://cdcvs.fnal.gov/redmine/projects/art.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, **Introduction to Algorithms, Second Edition**, MIT Press and McGraw-Hill, 2001.Section 16.3, pp. 385–392.